Izrada 2D računalne akcijsko-avanturističke igre tipa Metroidvania u programskom alatu Unity

Weisser, Fran

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Fran Weisser

IZRADA AVANTURISTIČKE IGRE METROIDVANIA ŽANRA U PROGRAMSKOM OKRUŽENJU UNITY

ZAVRŠNI RAD

Varaždin, 2021.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Fran Weisser

Matični broj: 0016131908

Studij: Poslovni sustavi

IZRADA AVANTURISTIČKE IGRE METROIDVANIA ŽANRA U PROGRAMSKOM OKRUŽENJU UNITY

ZAVRŠNI RAD

Mentor:

Doc. dr. sc. Mario Konecki

Varaždin, rujan 2021.

Fran Weisser

Izjava o izvornosti

Izjavljujem da je moj završni/diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor/Autorica potvrdio/potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Opis korištenja Game Engine aplikacije Unity u svrhu kreiranja prototipa 2D Metroidvania igre. Za početak objašnjenje pojmova bitnih za rad poput imena Metroidvania, njegovog podrijetla i tradicije. Kratka povijest Game Engine aplikacija i usporedbe najpopularnijih. Nakon toga se konkretna tema ovog rada - kreiranje glavnog lika, programiranje njegovih sposobnosti u interakciji sa svijetom, Object Pooling metode u svrhu instanciranja projektila po potrebi, programiranje pomičnih platforma, jednostavna umjetna inteligencija drugih likova u igri te stvaranje sustava kolizija i uništavanja neprijatelja u slučaju kolizije s određenim objektima. Stvaranje razine u kojoj će se sve odvijati, odvajanje na sekcije koje tek nakon određenog dijela postaju dostupne i završni neprijatelj nakon kojeg igra završava.

Ključne riječi: Video igra, Metroidvania, Unity, Game Engine, C#, proces razvoja video igre, prototip igre, igra u dvije dimenzije

Sadržaj

1 Uvod

Želja za stvaranjem video igara je danas vrlo često razlog studenata koji upisuju bilo koji fakultet koji ima veze s informatikom, a tako je definitivno bilo sa mnom. Za razliku od prošlih, današnje generacije su odrastale uz veliki pristup video igrama najčešće kroz svoj cijeli život. Naravno, nekima video igre ne predstavljaju ništa previše zanimljivo, ali neki u njima vide cijeli novi svijet koji nikad ne bi mogli pronaći van računala. Danas pogotovo uz sve nove mogućnosti koje nude aplikacije kreirane baš za tu svrhu takve ideje postaju izvedive za sve koji odluče uložiti malo truda. Zbog tog razloga je tema ovog rada ovo što je; baciti se u učenje novog i nepoznatog aplikacijskog okruženja u svrhu kreiranja svoje, makar i zasad vrlo jednostavne i predvidive video igre. U tu svrhu je kao što se vidi u naslovu korišteno Unity aplikacijsko okruženje koje sadrži veliku većinu mogućnosti potrebnih za kreaciju video igre.

2 Metode i tehnike rada

Za izradu ovog prototipa igre, a samim time i za pisanje rada je bilo potrebno koristiti se raznim izvorima informacija, ali su sve bile pronađene preko interneta i to najviše preko Google-a i YouTubea koji su prepuni informacija o temi ovog rada. Naravno, tu je najkorisnija bila dokumentacija samog programa u kojem se najviše dijela izrade odvijalo – Unity, gdje su opisane primjene i načini korištenja svih funkcija programa. Uz to bilo je i puno posjećivanja Unity foruma gdje bi često bila ponuđena rješenja na traženi problem zbog toga što su problemi prisutni u kreiranju video igre često univerzalni.

S obzirom na to da je programski jezik C# najčešće ključni dio rada u Unity programskom okruženju, aplikacija pogodna za njega je također bio bitan dio izrade video igre. Tijekom izrade projekta dijelom je bio korištena aplikacija Visual Studio 2019 a dijelom Sublime koji je unatoč manjoj funkcionalnosti znatno brži.

3 Video igre

Prije nego što se krene na glavnu temu rada, bit će rečeno nešto općenito o industriji video igara i podlozi za ovaj rad.

3.1 Industrija video igara

Video igre – najnovija i najkompleksnija vrsta umjetnosti koja ujedno nudi i bolji bijeg od stvarnosti od ičega prije njih. Tijekom zadnjih nekoliko godina industrija video igara je po zaradi pretekla industriju filmova i glazbe zajedno. Još kad se uzme u obzir prošla godina zatvaranja vanjskih objekata i prečestih zabrana uopće izlazaka na ulicu nije ni čudno što video igre i dalje postaju sve popularnije.

U našoj kulturi je dosad često bilo u modi ismijavati video igre, osuđivati one koji uživaju u njima, smatrati ih dječjima te samo pokušajem bijega od stvarnosti, baš kao što je to bilo i s filmovima nekoliko generacija prije. Ali sad polako dolazimo u vrijeme kad oni koji su igrali video igre na početku postaju dominantna generacija, vrijeme kad više ne postoje oni koji bi osuđivali video igre zato što su ih većina ljudi sadašnje generacija odrastali s njima. Samim time postaju napokon odobravanije od strane većine unatoč još uvijek čestim pokušajima da se njihova kulturološka vrijednost smanji. Uz to naravno, profit koji industrija ima danas je nenadmašiv. Zbog svega toga, video igre polako počinju imati sve bitniju ulogu u životu svih nas, koja je ponekad pozitivna a ponekad negativna.

Kako god bilo, video igre postižu nove razine uživljenosti, gdje više nije potrebno biti samo promatrač kao u primjerice filmu, već igrač postaje dio tog svijeta koji je virtualno stvoren. Potencijal koji igre imaju za budućnost je za nas danas nezamisliv, načini na koje će se razvijati dalje nikako ne možemo pretpostaviti jer kreativnost koju video igre omogućuju pogotovo uz programska okruženja poput Unity i Unreal Engine se povećavaju iz dana u dan.

Unatoč mainstream industriji koja stvara samo ono što će se prodavati, koja nažalost i dobiva većinu profita, manji timovi su oni koji tjeraju industriju naprijed, isprobavaju nove stvari, dolaze do dosad neviđenih kreativnih rješenja, novih koncepata koji pružaju iskustva koja se ne mogu proživjeti nigdje van svijeta video igre. Tako da, možemo pretpostaviti da nas iz ove industrije u budućnosti čekaju još zanimljivije stvari.

3.2 Metroidvania igra

Ime koje dolazi od spoja imena dvaju mega-popularnih igara 80-tih Castlevania i Metroid. One su obje imali ovaj specifični način razvoja svog svijeta zbog čega su postale predstavnici ovog tipa igara koje se i dan danas kreiraju.

Dakle, kao što već znamo, Metroidvania igra je 2D platformer poput primjerice igre Super Mario, ali ono što je čini drugačijom od te igre je veliki razgranati svijet koji je otvoren za istraživanje. To je svijet koji se širi u dvije osi (x, y) ili bolje rečeno u dvije dimenzije – lijevo-desno, gore-dolje. Zbog toga mapa tog svijeta koju te igre uvijek posjeduju nalikuje prikazima mravljih tunela u zemlji koje smo mogli ponekad vidjeti u knjigama biologije. Ali makar je svijet uvijek otvoren za istraživanje ne znači da ga se odmah može cijelog istražiti, tako se gubi na znatiželji i misteriju svijeta koji je stvoren.

Tako da uz veliki razgranati svijet, druga velika značajka Metroidvania igre je dobivanje novih sposobnosti koje igraču omogućuju da dosegne nove dosad nedostupne dijelove svijeta makar je put prema njima bio tehnički otvoren. To može biti primjerice sposobnost dvostrukog skoka, prolaženja kroz određenu vrstu prepreke ili lebdenja. Kako god bilo, uz nove sposobnosti novi dijelovi mape se „otključavaju“ igraču, tako da daljnje istraživanje postaje moguće.

Ovo dosad su bili mehanički opisi Metroidvania igre koji su dakako bitni, ali ono što zapravo čini ovu vrstu igre posebnom je osjećaj koji igrači dobiju tijekom igranja. Taj osjećaj rijetko koji tip igre može dostići u ovoj mjeri jer sama građa ovog tipa se zasniva na tome – misterioznost.

Jednom kad igrač uđe u novi svijet on polako ući početne sposobnosti i privikava se na igru. Najčešće nema putokaza pa je igrač prisiljen istraživati. U tom istraživanju igrač dolazi na razne prepreke i nagrade, borba s početnim protivnicima predstavlja zanimljivost, ali sve je to dosad viđeno. Ali onda dođe do dijela gdje ne može dalje – ili je skok potreban prevelik, ili je prisutna prepreka o kojoj se dosad ništa ne zna, i tu igrač postaje znatiželjan.

Kad jednom zapravo nađe pravi put, pobijedi glavnog neprijatelja na ovoj razini i krene dalje taj put koji nije mogao istražiti ostaje zabilježen u glavi. Takvih putova je najvjerojatnije susreo više puta, a svaki od njih može voditi u novi dio svijeta s novim misterijima, nagradama i protivnicima.

A možda tek nakon još nekoliko sati istraživanja on dobiva mogućnost da vidi što se zapravo nalazi iza tog slijepog puta, i onda nakon istraživanja tog puta možda susretne još nekoliko putova kojima ne može dalje. Na ovaj način se svijet polako razgranava u sve strane njegove dvije dimenzije, postaje sve širi i ide najčešće sve niže, sve dok jednom cijeli svijet nije istražen, kad je već igrajući lik pun svakojakih sposobnosti zbog kojih više nema problema s ijednim dijelom svijeta.

Sve opisano u igraču budi specifičan osjećaj avanture koji se može sresti rijetko gdje. Nijedna druga grana umjetnosti nije u mogućnosti postići toliku uživljenost kao video igre, a rijetko koja igra postiže ovako nešto, čemu i svjedoči činjenica da su i dan danas, 30 godina nakon izlaska prvih Metroidvania igara, još uvijek kreiraju igre ovog žanra.

3.3 Game Engine

Godinama video igre su bile rađene od nule, drugim riječima prije nego što bi se počelo uopće kreirati igru bilo je potrebno dizajnirati okruženje u kojem će igra moći biti kreirana. Taj proces je bio dugotrajan i mukotrpan posao gdje bi se tražili načini za optimalno korištenje hardvera da bi kasnije bilo što manje problema s primjerice zauzećem memorije a u isto vrijeme da igra može imati funkcionalnosti koje su u planu. U to vrijeme, dakle na početku, vrlo malo koda je moglo biti korišteno drugi put, tako da bi se najčešće sve nakon završetka igre odbacilo.

Tijekom devedesetih taj proces se pokušavao što više skratiti, tako da su se sve više počeli koristiti dijelovi prijašnjih igara kako bi se stvarale nove. Oni su se uskoro počeli licencirati kao posebni programi koji bi se mogli koristiti opet i opet u kreiranju novih igara, jer većina igara se baziralo na vrlo sličnim konceptima.

Iz toga su danas nastala nevjerojatno kompleksna programska okruženja koja nazivamo Game Engine koja u sebi sadrže nizove funkcionalnosti, mogućnosti i opcija sve u svrhu olakšavanja procesa izrade video igara. Primjerice sadrže opcije za 3D/2D modeliranje, rad s animacijama, korištenje raznih ustaljenih „game“ objekata, i opciju za pisanje koda.

Danas su daleko najpoznatiji već spomenuti Unreal Engine i Unity. Naravno, oba služe za istu stvar, ali na malo drugačiji način. Svaki ima svoje prednosti i mane, ali su oba kompleksni programi sposobni za mnoge stvari. Osim njih, koji su dosta robusni programi (Unreal zahtjeva 15GB memorije) postoje i neki manji a također kvalitetni alati od kojih bi se mogao izdvojiti Godot. Sa svojih nekoliko MB spreman je za rad nakon nekoliko sekundi. Unatoč tome što možda nije sposoban primjerice stvoriti toliko realistično okruženje poput Unreal Engine, sve više manjih timova se koristi njime zbog svoje visoke prilagođenosti manjim projektima i njegove open source prirode.

No, i dalje je Unreal Engine vjerojatno najpoznatiji od svih, s obzirom na svoju dugu povijest. Sposoban za nevjerojatno realistične prikaze grafike, dugački niz opcija od kojih je čak većina moguća za izvesti bez posebnog znanja programiranja zbog kao prvo opcija koje su uvedene kako bi se što više pojednostavio proces stvaranja, a i zbog velike zajednice koja se koristi njime koja već nudi rješenja za većinu problema koji mogu postojati za novog kreatora. Koristi se najčešće za velike projekte, visoko-budžetne igre kojima je cilj imati što

realističniju grafiku. Uz njih naravno da ga koriste i manji timovi, jer naravno sposoban za sve. Ali zbog svoje prvotne svrhe koja je bila stvoriti igru Unreal Tournament koja je FPS (First person shooter) igra, u njemu postoji malo ali i postojano naginjanje prema izradi FPS igara. Zbog toga su za neke druge žanrove igara, pogotovo ako nije cilj što realističnija grafika, primjerice Unity ili Godot se često preporučaju kao bolji izbor.

3.4 Unity

Za razliku od Unreala, koji je stvoren za određenu igru, Unity je napravljen bez cilja da se u njemu napravi igra. Glavna svrha mu je bila učiniti game development pristupačnijim većem broju ljudi. S obzirom na to da je pokrenut dosta rano (2005.) a i dan danas se i dalje unapređuje konstantno, Unity je danas sposoban za ogroman broj opcija. Uz to, besplatan je za sve koji se žele njime koristiti dok god nemaju prevelike prihode od igara kreiranih u njemu, u kojem slučaju je potrebno platiti za Plus, Pro ili Enterprise verziju. Zbog svega toga, Unity je vrlo popularan alat kod indie developera.

Makar je sposoban za vrlo realističnu grafiku, u tom dijelu i dalje malo zaostaje za Unreal Engine. Bez obzira na to, timovi se vrlo često odlučuju za Unity baš zbog tog razloga; realistična grafika sve više počinje biti cilj mainstream, visoko budžetnih tvrtki kojima je cilj prodati što više primjeraka igre, dok se manji timovi sve češće odlučuju za manje realističnu grafiku, na primjer crtanu rukom, ili pojednostavljene prikaze koji zrače originalnosti i stvaraju posebnu atmosferu. Kao primjere tu možemo navesti neke vrlo popularne igre iz posljednjih nekoliko godina koje su sve dostigle nove razine inovativnost; i što se tiče grafičkog stila ali i same mehanike igre i svijeta kojeg su stvorile.

Firewatch bi mogao poslužiti kao primjer prekrasnog grafičkog stila igre s inače dosta jednostavnim mehanikama, što se u svijetu igara danas često naziva simulatorom hodanja. Bez obzira na to, zbog vrlo zanimljivih likova, emotivne priče i već spomenutog predivnog stila vrlo pojednostavljene grafike nikad ne postaje naporna.

I za kraj možemo spomenuti igru koja je prva ponovno oživjela u to vrijeme pomalo zastarjeli žanr Metroidvanie – Hollow Knight. Također kreiran uz Unity, Hollow Knight je unatoč tome što tehnički ne čini ništa novo postigao nevjerojatan sklad mehanika igre, svijeta u kojem se odvija, priče koju je ispričao i glazbe. To sve zajedno kao cjelina stvara veliko i posebno iskustvo igranja koje se rijetko gdje može susresti.

Tako da, s obzirom na to da je ovo projekt izrade Metroidvania igre, dakle 2D igre dosta jednostavnih i već vrlo često utemeljenih mehanizama, bez potrebe za realizmom što se tiče grafike, Unity se čini kao savršeni izbor.

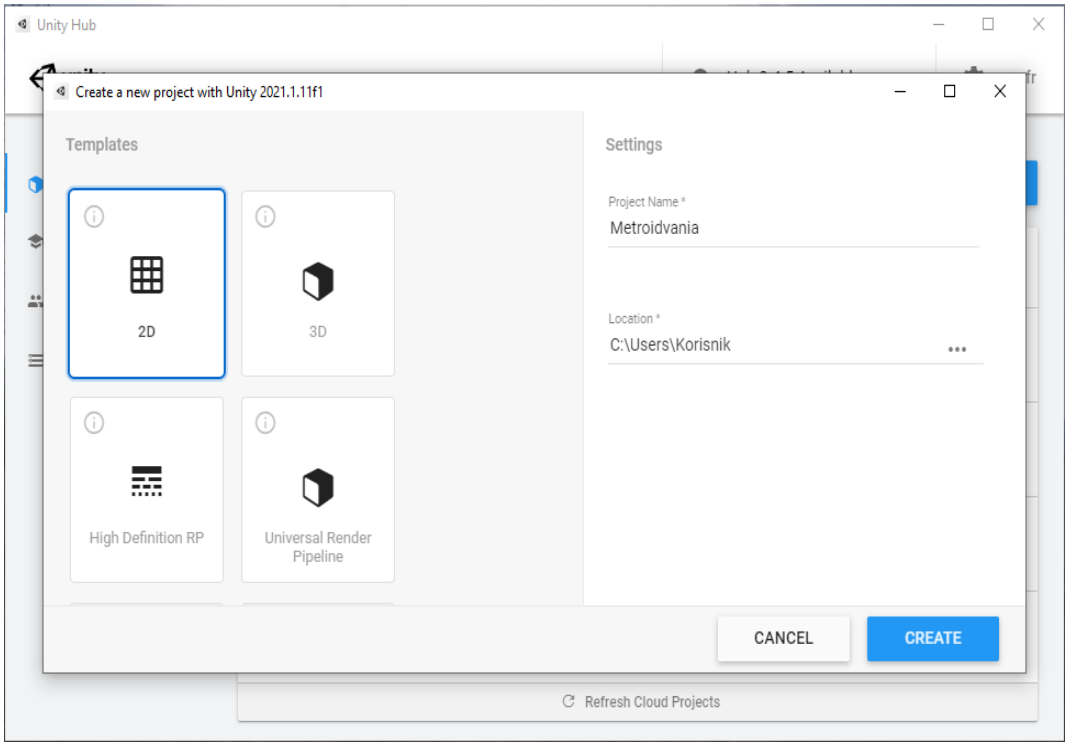
3.4.1 Instalacija

Prvo je dakako potrebno odabrati koju verziju Unity programskog okruženja želimo. Postoji nekoliko različitih verzija koje imaju minijaturne razlike, i odnose se najviše na to na koji način će Unity biti korišten. Tako postoji besplatna verzija koja ima većinu funkcionalnosti drugih verzija, samo što ne smije biti

Onda je potrebno preuzeti Unity, što danas najčešće ide preko drugog programa koji olakšava sve u vezi upravljanja različitim verzijama i projektima - Unity Hub. Uz Unity Hub preuzimanje verzije koju želimo je jako olakšano, samo je potrebno neko vrijeme da se program preuzme, a nakon toga on je odmah spreman za pokretanje.

3.4.2 Pokretanje

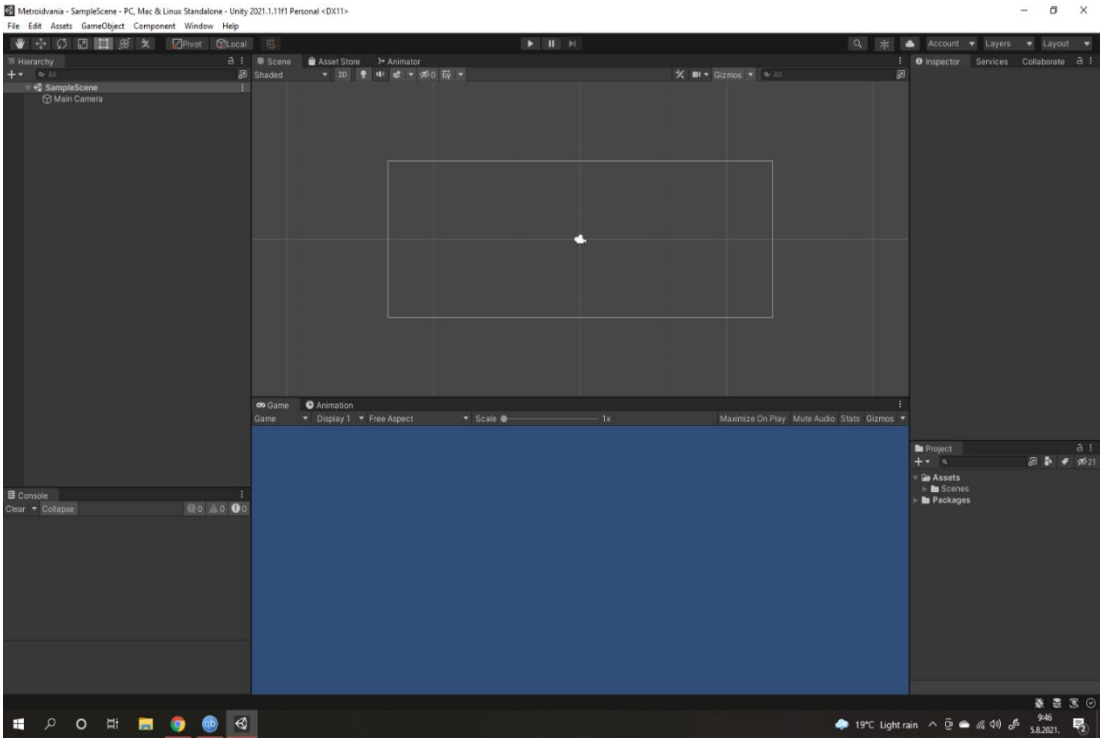
Kod prvotnog otvaranja potrebno je odabrati neke početne parametre kako bi se odredila vrsta projekta. Konkretno rečeno, Unity traži da odaberemo hoće li igra koju želimo stvarati biti dvodimenzionalna ili trodimenzionalna. Štogod odabrali ovdje ne čini praktički nikakvu razliku za dalje, samo postavlja prvotne postavke na 2D ili 3D što čini početak rada pogodniji za ono što želimo. S obzirom na to da je cilj ovog projekta izraditi Metroidvania igru u dvije dimenzije, tu su odabrane postavke za 2D.



Slika 1 Unity odabir vrste projekta

3.4.3 Početak korištenja sučelja

Nakon učitavanja novog projekta otvara se Unity. Prvo što hvata oko jer se nalazi na središtu ekrana je prozor u kojem vidimo svijet igre. S obzirom na to da još nemamo ništa, tako ni na ovom prozoru nema ničega osim pomoćnih linija koje olakšavaju snalaženje. Da smo odabrali 3D postavke, središte bi izgledalo identično kao u programima za 3D modeliranje poput Blendera, ali na 2D postavkama izgleda slično ali dakako bez treće dimenzije.



Slika 2 Unity početno sučelje

Osim središnjeg vidimo sa strane druge prozore od kojih svaki ima svoju svrhu, njihov raspored se naravno može proizvoljno uređivati, te sam ja odabrao raspored koji smatram da najbolje paše za potrebe ovog projekta.

S lijeve strane, prema rasporedu u aplikaciji koji se može proizvoljno uređivati se nalazi prozor Hierarchy gdje trenutno nema skoro ničega, ali ovdje će se nalaziti sve što se stavlja u svijet koji se gradi. Sve što postoji u tom svijetu Unity naziva objektima igre (Game Objects), koji se onda kombiniraju na razne način. Tako da njihovom interakcijom svijet igre postaje oživljen. Primjerice platforma je jedan objekt, a zid drugi, igrač treći a neprijatelj četvrti. Objekt koji se trenutno nalazi ovdje je samo kamera.

Dakle, to je prozor gdje gradimo svijet, a kamera je ono što njega spaja sa drugim prozorom koji se naziva Game View. Tamo više ne možemo graditi svijet, tamo ga možemo samo percipirati. U prvom prozoru mi smo u perspektivi kreatora igre, dok u Game View vidimo kako izgleda svijet iz perspektive igrača.

Ispod prozora Hierarchy vidimo konzolu. Tamo se ispisuju potencijalne greške u kodu, ili eventualne poruke koje nam sustav želi poručiti što se tiče aspekata programiranja igre, dakle konzola kao svugdje drugdje.

S druge strane su još dva prozora – Inspector i ispod njega Project. U inspektoru vidimo objekt koji imamo označen, njegove značajke, dodatne detalje i informacije preko kojih njime možemo upravljati pobliže. A u prozoru project se nalaze svi dijelovi projekta koje trenutno koristimo, gdje trenutno nema također ničega, ali gdje će se uskoro nalaziti sve skripte, animacije, slike i sve što odlučimo koristiti.

Dakle, uz sve to proces rada u aplikaciji se može lako razumjeti. Dodavanje objekata u prozor Hierarchy, oni se uređuju preko prozora Inspector, scena se gradi i prilagođava pomoću njih u središnjem prozoru, Project je tu gdje dodajemo nove skripte koje onda uređujemo te dodajemo na stvorene objekte preko inspektora, te pritiskom na strelicu na vrhu isprobavamo kako igra funkcionira u perspektivi igrača. Rad postaje poprilično intuitivan dosta brzo.

4 Stvaranje video igre

Kao što već znamo, stvaranje video igre zahtjeva puno vremena i truda, spajanje nevjerojatno puno različitih grana i računalne znanosti i umjetnosti. Tako da zahtijevaju objedinjavanje tehničkih dijelova poput programiranja, i grafičkih aspekata poput 3D modeliranja i animiranja u primjerice Blenderu ili 2D prikaza za što je pogodan Photoshop ili Krita. Uz to još je nužna i obrada zvuka korištenih u samoj igri ali i kreiranje glazbene podloge koja je također bitna za stvaranje ugođaja. I prije svega, potrebna je kreativna ideja za igru na kojoj sve drugo navedeno može stajati, bez čega ništa drugo nema nikakve svrhe.

Što se kreativnog dijela tiče tu je danas često i pisanje priče koja može biti ispričana na kreativne načine svojstvene igri u kojoj se odvija, ili što je vrlo često u većim komercijalnim projektima pisanje likova, događaja i radnje kao u filmovima. U tom slučaju na sve navedeno je potrebno dodati i standarde za snimanje filmova; od režiranja, filmografije i pisanja scenarija pa sve do glumljenja. Takvih igara se počelo u zadnjih 10ak godina proizvoditi sve više, tako da budžeti potrošeni na takve igre su enormni, te je to često neizvedivo za manje projekte. Ali opet, baš zbog toga današnja industrija komercijalnih igara je jako izgubila na kvaliteti, te je svaka igra stvorena na isti kalup.

Zbog svega toga je često nužna suradnja većeg broja ljudi, a za manje projekte suradnja barem nekoliko osoba bitno različitih pozadina gdje bi primjerice jedna osoba bila zadužena za tehničke aspekte igre a druga za kreativne. Naravno, postoje i slučajevi gdje je jedna osoba odradila sve navedeno (osim glume), kao primjerice igra koja danas ima kultni status - Undertale, ali takvi su slučajevi veoma rijetki, s time da zahtijevaju ogromnu raznovrsnost interesa i vještina jedne osobe. No, bez obzira na to, to je dakako izvedivo i danas je sve češće.

Kad se sve to uzme u obzir, postaje očito da čak i stvaranje male igre zahtjeva ogromne razine truda i vještina. Pogotovo igre Metroidvania žanra, koje su unatoč često jednostavnim i danas dosta poznatim tehnikama i dalje zahtijevaju izgradnju velikog svijeta, što naravno traži puno optimiziranja i truda.

Zbog tog razloga ovaj projekt će biti manje usredotočen na to da se stvori potpuno funkcionalna i optimizirana igra, već prototip igre koji bi se u budućnosti mogao proširiti da bi se stvorila prava Metroidvania igra. Tako primjerice neće biti velikog svijeta, u kojem će biti raspršene nove sposobnosti za igrača, već će se sve nalaziti u jednoj sekciji.

4.1 Proces kreiranja žanra Metroidvanie

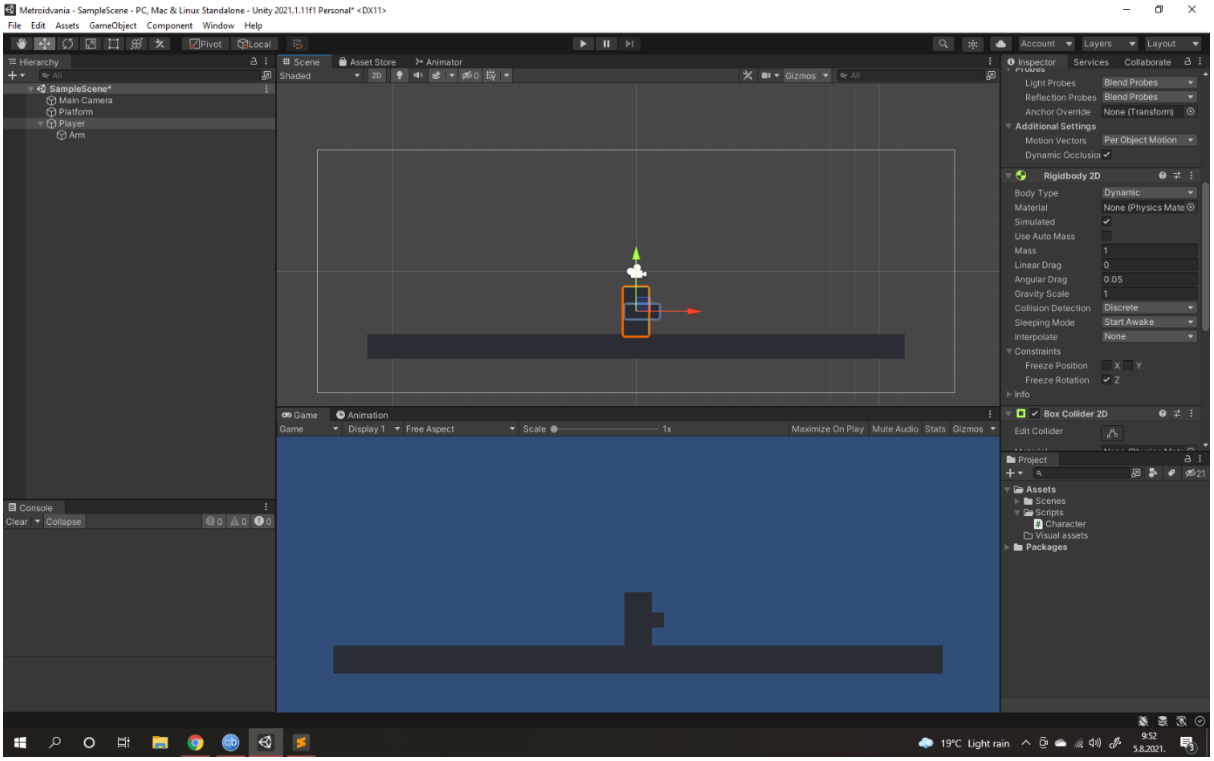
Za početak će biti prikazano kako stvoriti lika koji će biti pokretan od strane igrača. Njemu će biti dodane sve sposobnosti i mogućnosti kojima će tijekom igre imati pristup. Tu idu i klase koje odrađuju instanciranje i ispaljivanje projektila.

Što se dizajna lika tiče, u 2D igrama je često korištenje alata poput Photoshopa da bi se dizajnirali likovi i stvorile animacije pogodne za njih. Taj dio će u ovom projektu biti veliki dijelom preskočen zbog toga što je to još jedna dodatna grana vještina koja nije potpuno nužna, ali u isto vrijeme i zato što je tijekom kreiranja ovog prototipa došao do izražaja stil koji u koji se čini da vrlo lijepo pristaje igri. Stil gdje je cijeli svijet igre prikazivan u kombinacijama jednostavnih 2D oblika dostupnih u Unity.

Tako je primjerice lik kojeg igrač kontrolira na početku iz razloga da se dizajn ostavi za kasnije bio samo kocka sa „rukom“ (manji pravokutnik), ali uz lijepu kombinaciju boja takva dva jednostavna oblika su oživjela. Na taj način je rođen cijeli vizualni stil u kojem će ovaj prototip biti prikazan.

4.2 Stvaranje lika za upravljanje

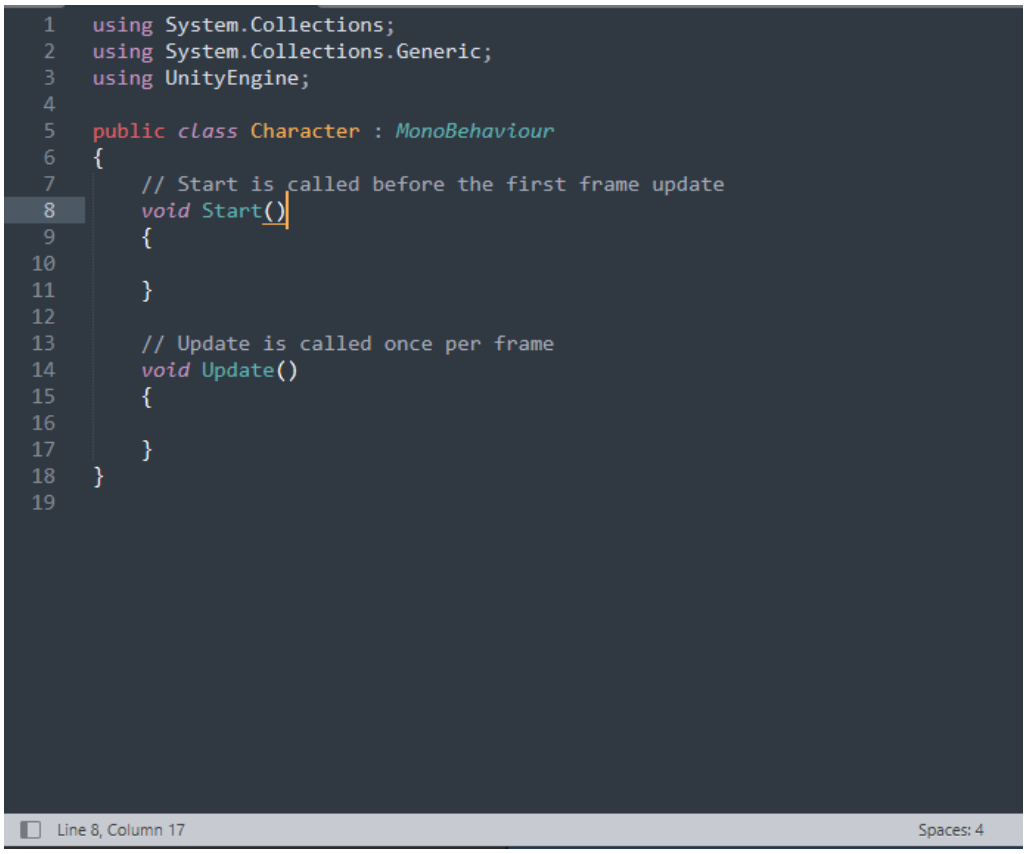
Kako bi stvorili lika koji se može kretati, skakati i ostatak sposobnosti koje su česte u Metroidvania igrama bit će potrebno nekoliko koraka. Za početak možemo stvoriti novi objekt koji će igrati ulogu lika dok još nemamo dizajn ili gotove animacije - pravokutnik će poslužiti. Na njega možemo staviti još jedan pravokutnik da vidimo u kojem smjeru je lik okrenut. Ispod njega možemo postaviti i jednu platformu kako bi imao na čemu stajati.



Slika 3 Početak rada u Unity

Preko prozora Inspector vidimo što trenutno postoji na našem liku. Prva stavka transform nam govori o lokaciji na kojoj se nalazi, njegovoj rotaciji i veličinu. Druga stavka sprite renderer je vizualan dio pravokutnika kojim se služimo. Uz njih potrebno je dodati i BoxCollider2D što pokazuje njegove granice, može se reći preko kojih komunicira sa ostatkom svijeta i RigidBody2D kako bi bio tretiran kao stvarno fizičko tijelo u Unityju, tamo se mogu i označiti ograničenja koja su pogodna za lika u igri– zamrzavanje rotacije na z osi. Uz to, što će biti potrebno za kasnije ali je svejedno postavka koja se može odmah postaviti, možemo stvoriti novi sloj koji će se zvati Player i pridružiti ga objektu Player. Isto vrijedi i za većinu drugih stavki koje se stvore poput platformi, neprijatelja ili zidova kako bi se olakšao rad s detekcijom Collidera kasnije.

Nakon toga kako bi s likom išta mogli učiniti potrebno je mu dodati i još jednu stavku – novu skriptu.



Slika 4 početni izgled Unity skripte

Otvaranjem nove skripte vidimo da se ona trenutno sastoji od dvije prazne metode: Start() i Update(). To su specifične metode za Unity koje dolaze zajedno s Unity.Engine, zajedno s još velikim brojem drugih metoda. Dakle, ono što je u Start() se pokreče čim se igra pokrene, dok ono što je pod Update() se pokreče ponovno za svaki novi Frame. Uz njega tu je i FixedUpdate() koji za razliku od Update() ne ovisi o broju FPS-a, te je zbog toga pogodniji za rad s fizikom. U kodu će često biti i jedan i drugi, primjerice metoda koja detektira ako je tipka pritisnuta će biti obavljana u Update(), dok će obavljanje primjerice skoka biti odrađeno u FixedUpdate().

Za početak možemo dodati novi namespace pod kojim će sve skripte ovog projekta biti - Metroidvania. S obzirom da je ovo samo generalna skripta koja služi za objedinjavanje drugih zasad neće sadržavati puno toga, a s vremenom će ovdje biti dodane metode koje ili provjeravaju stanja lika ili primjerice okreću lika u smjeru gdje treba biti okrenut. Ali zasad je dovoljno napraviti vezu s Rigidbody2D i Collider2D na objektu Player te obaviti inicijalizaciju na koju će se sve kasnije vezati što sve vidimo na slici. Bit će još jedna skripta nazvana Abilities koja će sadržavati sposobnosti igrača, koja je također potrebna da bi ih sve objedinila te zasad nema puno koda. Dalje će ukratko biti prikazane osnovne sposobnosti koje lik mora sadržavati i kako su odrađene u kodu, od kojih je prva i najosnovnija kretanje.

4.2.1 Kretanje lika

Kretanje će biti odrađeno u skripti nazvanoj HorizontalMovement. Uz Unity ova funkcionalnost je dosta jednostavna za napraviti. U sljedećem isječku koda vidimo metodu koja detektira kretanje u lijevo i desno koje su u Unity postavljene automatski na strelice lijevo-desno i na slova A i D. Ona se postavlja u Update().

```
protected virtual bool MovementPressed()
        {
            if(Input.GetAxis("Horizontal") != 0)
            {
                horizontalInput = Input.GetAxis("Horizontal");
                return true;
            }
            else
                return false;
        }
```

A sljedeće ostaje stvoriti metodu koja će zapravo utjecati na tijelo tako da se ono kreče uz te tipke koje su navedene u prošloj metodi. To je moguće na ovaj način:

```
protected virtual void Movement()
        {
            if(MovementPressed())
            {
                walkSpeed = horizontalInput * speed * Time.deltaTime;
            }
            else
            {
                walkSpeed = 0;
            }
            CheckDirection();
            rb.velocity = new Vector2(walkSpeed, rb.velocity.y);
        }
```

Dakle, dok vrijedi prošla metoda prethodno stvorena varijabla brzina hoda (walkSpeed) se umnožava s drugom proizvoljnom varijablom i u ovisnosti s vremenom (Time.deltaTime). CheckDirection() je metoda napisana koja provjerava smjer u kojem se igrač kreče tako da brzinu podešava po potrebi na pozitivnu ili negativnu vrijednost. Zadnja linija prikazanog koda (rb.velocity) daje tijelu silu jačine walkSpeed varijable na X osi uz Unity Vector2 varijablu, koja sadržava dvije vrijednosti što korespondira položaju tijela u 2D prostoru.

4.2.2 Flip metoda

Uz CheckDirection() ovdje je dosta bitna i metoda koja će promijeniti smjer u kojem je igrač okrenut onda kad se kreče u suprotnom smjeru, unatoč tome što je tako nešto manje bitno u potrebama ovako jednostavne grafike svejedno je implementirano uz metodu Flip().

```
protected virtual void Flip()
            {
                if(isFacingLeft)
                {
                    transform.localScale = facingLeft;
                }
                else
                {
                    transform.localScale = new Vector3(-
transform.localScale.x, transform.localScale.y, transform.localScale.z);
                }
            }
```

4.2.3 Trčanje (Boost)

Trčanje je česti dio velikog broja video igara, tako će i ovdje biti implementirano ali na malo drugačiji način. Umjesto trčanja funkcionirat će više kao dodatni pogon zbog koje se lik može kretati brže.

Uz novu varijablu sprint koja je 1 dok tipka LeftShift nije pritisnuta a 2 kad je, te Unity metodu koja to provjerava – GetKey(KeyCode.LeftShift), trčanje je vrlo lako izvedeno uz dodavanje te nove varijable u metodu Movement.

4.2.4 Skok

Kako bi se implementira skok potrebno je malo više koraka. Za početak je potrebno detektirati kad je pritisnuta tipka Space, ali ne metodom GetKey koja je dosad bila korištena, već metodom GetKeyDown koja detektira samo ako je tipka pritisnuta. U suprotnom bi tijelo dobivalo silu na prema gore dok god igrač drži tipku Space što prestaje biti skok.

Sljedeće je potrebno detektirati kad lik stoji na zemlji, jer samo dok stoji na zemlji on može skakati. To će biti obavljeno metodom CollisionCheck() koja će biti stavljena u klasu Character, jer će biti korištena za veliki broj namjena.

```
protected virtual bool CollisionCheck(Vector2 direction, float distance,
LayerMask collision)
{
        RaycastHit2D[] hits= new RaycastHit2D[10];
        int numHits = col.Cast(direction, hits, distance);
        for(int i = 0; i < numHits; i++)
        {
            if((1 << hits[i].collider.gameObject.layer & collision) != 0)
                {
                    return true;
                }
            } return false;
}
```

Ovdje je korištena metoda RaycastHit2D koja usmjerava zraku u smjeru u kojem je traženo (Vector2 direction) i detektira traženi sloj (LayerMask) ako je player.Collider2D s njime u kontaktu na daljini koja joj se zada (distance). Uz nju detekcija tla je jednostavna, odrađena je u metodi nazvanoj GroundCheck().

```
protected virtual void GroundCheck()
        {
            if(CollisionCheck(Vector2.down, distanceToCollider,
collisionLayer))
            {
                character.isGrounded = true;
                numberOfJumpsLeft = maxJumps;
            }
            else
            {
                character.isGrounded = false;
            }
        }
```

4.2.5 Lebdenje

Još jedna sposobnost koju igrač ima je lebdenje kako bi usporio brzinu pada. Na taj način ima mogućnost prelaženja većih udaljenosti u skoku. Za nju je potrebno prvo provjeriti da li je lik trenutno u zraku, a to će biti obavljeno metodom Falling() u Character klasi:

```
protected virtual bool Falling(float velocity)
            {
                if(!isGrounded && rb.velocity.y < velocity)
                {
                    isFalling = true;
                    return true;
                }
                else
                    return false;
            }
```

A nakon toga je dovoljno brzinu pada (velocity.y) podijeliti s brojem kojim želimo ovisno o tome koliko želimo usporiti pad.

4.2.6 Dash sposobnost

Sljedeća sposobnost je možda najpopularnija u igrama ovog tipa. Službeni naziv korišten svuda u svijetu video igara joj je dash, a sastoji se od kratkog i brzog zaleta lika kojeg igramo u smjeru u kojem želimo. Korišten je vrlo često u svrhu izbjegavanja neprijateljskih napada ili produžavanja skoka kako bi se došlo na udaljenije platforme.

Implementacija njega je malo kompleksniji zadatak od prijašnjih te je ovdje izvedena uz više metoda. Prva metoda Dashing() provjerava ako je pritisnuta tipka, a ako je; započinje oduzimati vrijeme koje je dopušteno da Dash traje, blokira sve pozive za kretanje, proglašava varijablu isDashing istinitom i započinje suprogram (Couroutine) nazvann FinishedDashing.

```
protected virtual void Dashing()
{   if(input.DashPressed() && canDash)
    {       dashCountDown = dashCooldownTime;
            character.isDashing = true;
            movement.enabled = false;
            StartCoroutine(FinishedDashing());
      }
}
```

Coroutine je način u Unity da se da zada pauza određenog vremena prije izvršavanja sljedeće naredbe. U Unityiju ju se implementira uz tzv. IEnumerator koji je u ovom slučaju pokrenut nakon što istekne vrijeme zadano trajanju izvršavanja Dash sposobnosti u igri, te on vraća varijablu isDashing na neistinu te ponovno dopušta horizontalno kretanje.

Osim ove dvije metode tu je i DashMode() koja zapravo izvršava Dash uz pomoć AddForce u željenom smjeru te isključuje koliziju za određeni sloj uz pomoć metode DashCollision. Uz navedeno je moguće napraviti sloj kroz koji će igrač moći proći samo uz pomoć Dash sposobnosti.

I zadnja metoda ResetDashCounter() služi za to da nakon izvršavanja Dash sposobnosti postoji određena pauza prije ponovnog korištenja.

4.2.7 Pomicanje kamere zajedno s likom

Kako bi se kamera micala zajedno s likom dovoljno ju je učiniti djetetom objekta Player. Moguće je i samo napisati u skripti koja joj pripada transformaciju uvijek na mjesto igrača, ali na ova dva načina ona izgleda jako čvrsto na igraču i ne djeluje glatko.

Zbog toga je korištena metoda Vector3.Lerp koja matematički prilagođava poziciju na igrača ali ne toliko oštro, već ima okvir u kojem može kretati slobodno. Na ovaj način kamera daje ljepše i glađe. Metroidvanie pogotovo ovog stila gdje se cilja uz pomoć miša često nude još bolje načine prijelaza kamera, te prilagođavanje poziciji miša, ali za potrebe ovog rada to nije previše nužno s time da već i ova mala korekcija kamera izgleda lijepo.

4.3 Borba

Bitna značajka Metroidvania igara je borba s neprijateljima. Ta borba se može odvijati na blizini (Melee) ili na daljinu uz pomoć mehanika pucanja. U igri čiji se razvoj pokazuje u ovom radu će biti implementirana druga opcija na liku kojeg igrač pokreče, dok će za borbu na blizinu biti sposobni samo neprijatelji.

Postoji puno različitih načina za implementaciju ciljanja i pucanja. Možda najjednostavniji bi bio uz pomoć naredbe koja je već bila spomenuta kao dio Unity.Engine – RaycastHit2D. Dakle, naredba stvori zraku na željenom mjestu i smjeru koja detektira sve što je u tom trenutku na njenom putu. Na taj način se može detektirati kolizija na željenom mjestu. Zraka se može i animirati tako i tehnika je gotova. To je način koji funkcionira u nekim igrama, poput igara FPS tipa gdje je potrebno da metci budu detektirani istog trenutka kad je i opaljen. Ali za druge igre, poput 2D platformera, pogodnije je a i zanimljivije da taj projektil ima stvarnu putanju po kojoj se kreče u stvarnom vremenu.

Zbog toga će ovdje biti prikazan taj drugi pristup, koji se također može implementirati na bezbroj načina, ali se svi temelje na istim stvarima: projektil koji je stvoren u trenutku pritiska gumba koji se kreče u željenom smjeru. Taj pristup je malo kompliciraniji, te će zahtijevati više koraka da bi se implementirao na što optimalniji način.

4.3.1 Rotiranje ruke u smjeru miša

Prvi korak bi bio odabrati na koji način će se odabirati smjer ispaljivanja projektila. Postoji mogućnost koja dosta česta u igrama ovog tipa ispucavanja horizontalno u smjeru u kojem je igrač okrenut, a druga manje prisutna mogućnost je ispucavanje u smjeru pozicije gdje se nalazi miš. Pošto se za prototip ove igre čini daleko pogodnija druga mogućnost, potrebno je prvo stvoriti način da se ruka igrača rotira u smjeru miša iz koje će se onda instancirati projektili za ispaljivanje.

A kako bi ju rotirali prvo je potrebno napraviti novi prazni objekt otprilike na mjestu ramena, a ruku učiniti djetetom tog objekta. Na taj način se rotiranjem novog objekta ruka rotira na način na koji je potrebno a ne oko svoje osi. Nakon toga metoda FollowMouse() služi za rotiranje ruke.

```
protected virtual void FollowMouse()
{
   Vector2 difference = Camera.main.ScreenToWorldPoint(Input.mousePosition)
   - transform.position;
   difference.Normalize();
   float rotationZ = Mathf.Atan2(difference.y, difference.x) *
   Mathf.Rad2Deg;
   if(player.transform.localScale.x > 0)
               {
                   transform.rotation = Quaternion.Euler(0, 0, rotationZ);
               }
    else
               {
                   transform.rotation = Quaternion.Euler(180, 180,
rotationZ);
               }
           }
```

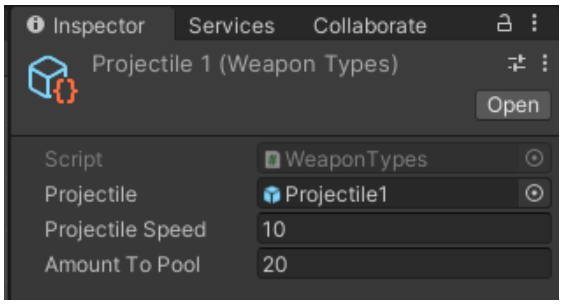
Uz pomoć Unity naredbi (Camera.main.ScreenToWorldPoint(Input.mousePosition) moguće je stvoriti objekt u igri koji će posjedovati svoju Vector2 varijablu na mjestu pokazivača, drugim riječima lokaciju miša. Uz pomoć njega i malo matematike je moguće učiniti da se novi objekt rotira u smjeru miša te da se obrne na drugu stranu kad je igrač okrenut.

4.3.2 Skriptirani objekti

Prvi dio bi bio stvoriti tzv. skriptirani objekt (Scriptable object) koji će služiti kao projektil koji se ispaljuje. Potrebno je stvoriti njegovu jednu instancu te ju prebaciti iz Hierarchy prozora u Project. Na taj način se stvara od objekta Prefab, zbog čega je nakon toga napisan plavom bojom, koji se onda može stvoriti po želji identičan takav kakav je pridružen prozoru Project.

Nova skripta WeaponTypes će poslužiti za stvaranje i upravljanje skriptiranim objektima. Prva linija prikazana na slici služi kako bi se stvorila mapa koja će sadržavati skriptirane objekte koji se onda mogu po volji stvarati novi. Kao što vidimo, svaki navedeni objekt zasad ima stavku gdje se pridružuje potrebni objekt (projektil) , brzinu i broj instanci koji se trebaju stvoriti uz pomoć sljedeće skripte Object Pooler. Tako se mogu dobiti različite vrste projektila bez previše komplikacija, zbog čega se i klasa zove WeaponTypes. Unatoč tome u ovom projektu će biti samo jedan tip projektila makar je prvotno plan bio implementirati više vrsta što se vidi i u kodu.

Kako bi se stvorio projektil od kojeg je potrebno stvoriti skriptirani objekt dovoljno je na prazni objekt staviti sliku metka koju želimo, i dodati mu i prilagoditi Collider2D i Rigidbody2D.



Slika 5 Skriptirani objekt projektil

4.3.3 Object Pooler

Drugi korak nije nužan ali je vrlo čest način implementiranja ove svrhe - stvaranje objekta koji će stvarati nove instance projektila (skriptirane objekte) po potrebi. Ovdje je stvoren novi objekt nazvan ObjectPooler koji stvara broj instanci projektila koliko je zadano skriptiranom objektu (Projektilu) i postavlja ih prvo na neaktivne te ih po potrebi postavlja na aktivne na traženo vrijeme, i nakon čega ih vraća u neaktivne.

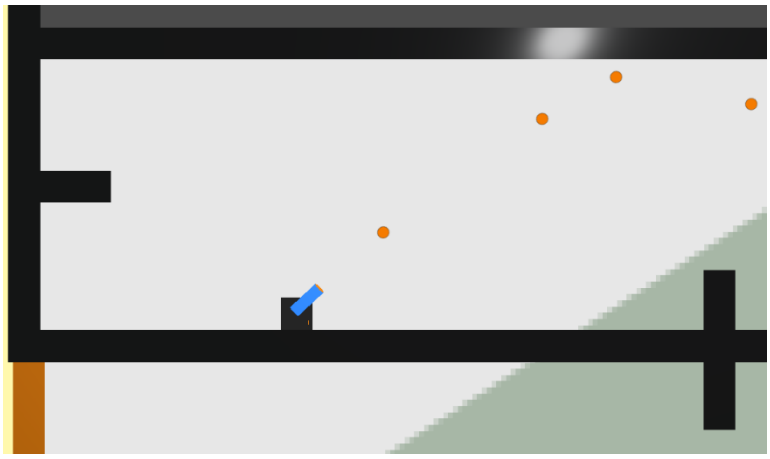
```
public void CreatePool(WeaponTypes weapon, List<GameObject> currentPool,
GameObject projectileParentFolder)
{
    for(int i = 0; i<weapon.amountToPool; i++)
    {
      currentItem = Instantiate(weapon.projectile);
      currentItem.SetActive(false);
      currentPool.Add(currentItem);
      currentItem.transform.SetParent(projectileParentFolder.transform);
    }
    projectileParentFolder.name = weapon.name;
}
```

Na ovaj način se znatno štedi memorija jer nema potrebe za kontinuiranim stvaranjem novih objekata, jer se uvijek isti objekti ponovno koriste na kratko vrijeme nakon čega opet nestaju.

4.3.4 Ispucavanje projektila

U skripti nazvanoj Weapon, koja je dio potomstva Abilities klase se obavlja instanciranje projektila onda kad je to potrebno (na pritisak lijeve tipke miša). U tom trenutku je projektil koji je već stvoren preko Object Poolera ali neaktivan postavljen na aktivno. Stvara ga na praznom objektu koji je nanovo stvoren na kraju ruke na liku, tako da preuzima rotaciju i poziciju tog ruke.

Nakon toga još jedna skripta Projectile postavlja vrijeme u kojem je projektil aktivan, te ga ispaljuje u traženom smjeru i brzinom, što je oboje zadano u skripti WeaponTypes.



Slika 6 Ispaljivanje projektila

4.4 Platforme

Kako bi svijet bio malo življi, te da bude manje monotono ako su samo neprijatelji prepreke, neke Metroidvanie imaju „platformerske“ motive. Najčešći bi dakako bili platforme koje se uništavaju kratko vrijeme nakon što igrač skoči na nju te platforma koja se pomiče. Osim njih naravno da postoji i mnoštvo drugih, ali ove dvije su kreirane ovdje.

4.4.1 Platforma koja se uništava

Ova platforma uz pomoć RaycastHit2D.BoxCast stvara pravokutnik koji detektira igrača nešto iznad collidera same platforme, ako metoda vrati istinu pokrenuto je oduzimanje vremena i nakon određenog broja sekundi platforma se pretvara iz Kinematic tijela u Dynamic zbog čega na nju počinje djelovati gravitacija u igri. Nakon određenog broja padanja ona je postavljena na neaktivno zbog čega nestaje. U posljednjoj metodi ResetPlatform() postaje ponovno aktivna vraćena na istu poziciju s početka i tijelo postavljeno na Kinematic.

Zbog toga što nije zamrznuta rotacija na Z osi platforma u padu može promijeniti rotaciju te nakon što je ponovno stvorena posjeduje rotaciju koju je imala dok je padala. Bug koji se lako može riješiti, ali je zbog efektnog izgleda ostavljen neriješen.

4.4.2 Platforma koja se pomiče

Zadane su dvije pozicije koje se navode u Inspectoru, uz njih tu su i dvije varijable moveForward i moveBack, te trenutna pozicija same platforme. Kombinacijom ovih elemenata i metodom Vector2.MoveTowards je postignuto da se platforma pomiče s jedne lokacije na drugu, i kad ondje stigne se vraća na prvu.

4.5 Neprijatelji

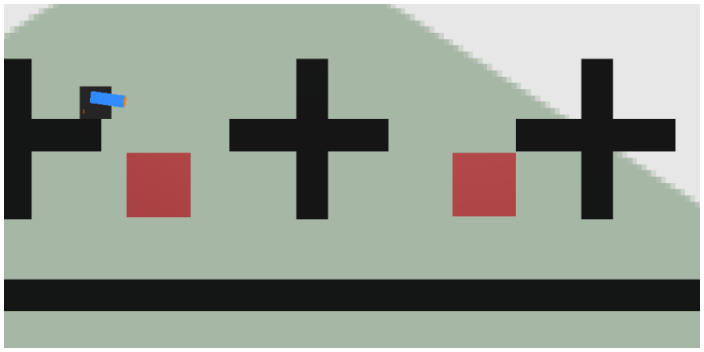
Ključan dio većine Metroidvania igara uz razgranati svijet je borba s neprijateljima. Njih inače ima mnoštvo različitih tipova, uglavnom nekoliko za određenu sekciju svijeta. Od slabijih, jednostavnijih, manjih koji ne predstavljaju izazov nego malenu prepreku u istraživanju svijeta te nemaju previše sposobnosti, pa sve do tzv. „boss“ neprijatelja koji imaju posebno razrađenu umjetnu inteligenciju s velikim brojem različitih sposobnosti. Borba s „bossevima“ i nadvladavanje njih je često ono što ostane u pamćenju igrača dugo nakon igranja igre, pogotovo ako su veliki izazov, kao što u Metroidvania igrama često jesu.

U ovom projektu, s obzirom na to da je ovo prototip igre, neće biti pravog boss neprijatelja, nego samo malo drugačiji neprijatelj od ostalih čiji poraz će značiti pobjedu nad igrom. Što se tiče ostalih neprijatelja, bit će nekoliko jednostavnijih tipova koji će biti raspršeni po svijetu. S obzirom na to da je svijet igre dizajniran bez tekstura ili korištenja crtanja ili 3D modeliranja, tako će i neprijatelji biti dizajnirani uz pomoć 2D objekata dostupnih u Unity.

Kod će biti odrađen uz pomoć klase EnemyCharacter koja ide na svakog neprijatelja te sadrži neke podatke koje svaki neprijatelj treba posjedovati poput mogućnosti da bude uništen ili da on uništi igrača. Uz to svaki tip će imati još barem jednu skriptu koja će se odnositi direktno na specifičnosti tog tipa.

4.5.1 Enemy1/1

Prvog neprijatelja će predstavljati jednostavan kvadrat koji će se kretati u jednom smjeru sve dok ne dodirne ili zid ili igrača, u kojem slučaju ga ošteti. Osim toga ovaj prvi tip nema previše razrađene umjetne inteligencije. Uz to neuništiv je za igrača kako bi predstavljao barem nekakav izazov. On se pokreče uz pomoć transform.position tako da se ne koristi fizikom, zbog toga ima Rigidbody postavljen na Kinematic.



Slika 7 Enemy1

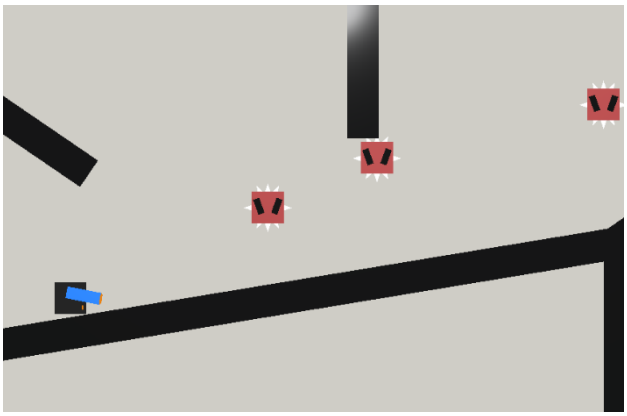
4.5.2 Enemy1/2

Drugi neprijatelj prvog tipa je isti kao i prvi samo što ima drugačiji način kretanja, umjesto da ide lijevo desno ovaj skače poput žabe. Njegovo kretanje pokreče sila (AddForce), zbog čega je potrebno promijeniti njegovo fizičko tijelo s Kinematic na Dynamic što je sve obavljeno u skripti. Oba neprijatelja prvog tipa su kreirani u jednoj skripti i biranje koji će objekt će biti koje vrste neprijatelja se odrađuje u Inspectoru. Oba služe više kao prepreke koje je potrebno izbjegavati a manje kao neprijatelj protiv kojeg je se potrebno boriti.

4.5.3 Enemy2

Drugi tip neprijatelja je također kvadar, ali će ovaj imati još dvije crte na sebi koje pokazuju njegovo negativno raspoloženje. Ali ovaj je vrlo mali kvadar koji leti, ima razrađeniju umjetnu inteligenciju te on detektira igrača uz pomoć Raycast2D u funkciji DetectPlayer onda kad uđe u krug koji naredba CircleCast stvara. Nakon što ga detektira on leti na njega brzinom kojom se podesi te ošteti igrača ako dođe do kolizije u funkciji FollowPlayer(). Ovaj neprijatelj će biti brz ali lak za uništiti, uz to najčešće ih se može vidjeti u većim grupama.

```
void DetectPlayer()
            RaycastHit2D hit;
            hit = Physics2D.CircleCast(col.bounds.center, radius,
Vector2.zero, 0, layer);
            if (hit)
            {
                playerFound = true;
                attack = true;
      }
```

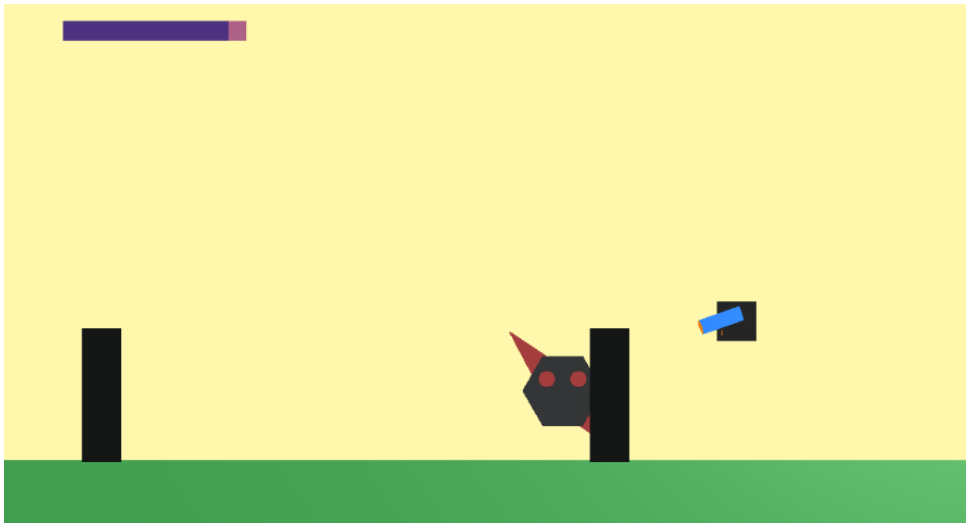


Slika 8 Enemy2

4.5.4 Enemy3

Treći tip neprijatelja je kombinacija prvog i drugog tipa s time da koristi i kretanje prvog tipa dok nije locirao igrača i funkcije DetectPlayer() i FollowPlayer() nakon čega ga počinje ganjati. Uz to dizajniran je tako da objekt koji strši iz njega koji se okreče jednom brzinom dok ne vidi igrača te dok ga vidi tom brzinom poduplanom što je implementirano uz GetChild() metodu koja pronalazi objekt koji nasljeđuje objekt neprijatelja te mu mijenja rotaciju.

```
protected virtual void FollowPlayer()
{
    if (playerFound)
    {
       Vector2 distanceToPlayer = (new Vector3(transform.position.x-2,
transform.position.y)-player.transform.position).normalized +
player.transform.position;
                transform.position =
Vector2.MoveTowards(transform.position, distanceToPlayer, runSpeed *
Time.deltaTime);
            }
        }
```



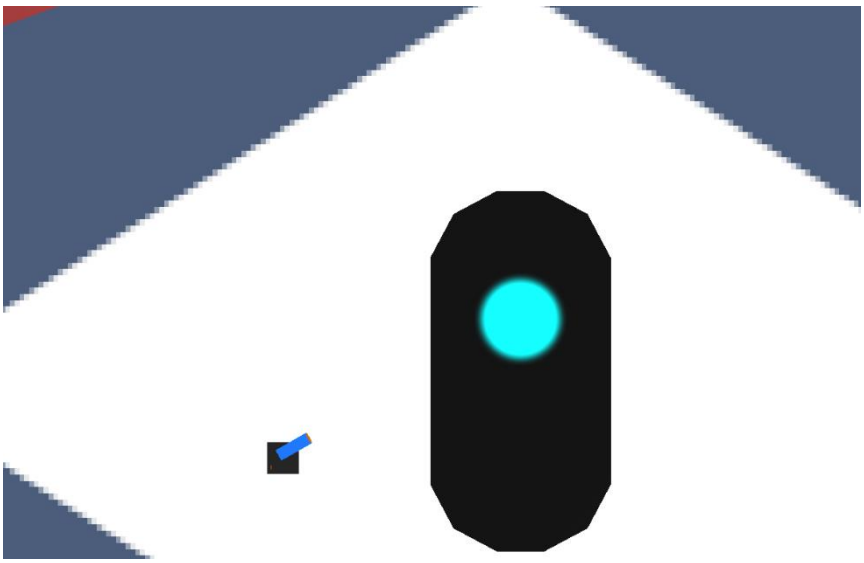
Slika 9 Enemy3 u svom prirodnom okruženju

4.5.5 Enemy4

Četvrti tip neprijatelja je boss ovog prototipa, dakle služi kao posljednja prepreka u prototipu igre nakon koje igra završava te se pojavljuje slika pobjede. Sistem prema kojem se pojavljuje ova slika će biti objašnjen u sljedećem poglavlju gdje je na isti način odrađena slika neuspjeha.

A sami neprijatelj je kao prvo u odnosu na ostale tipove znatno veći te zahtjeva više manevriranja i posjeduje veću količinu zdravlja. Uz to jedini ima mogućnost ispaljivanja projektila na istom principu kao i igrač. Uz skriptu ObjectPooler on stvara dvije vrste projektila te ih ispaljuje iz rotirajućih mjesta, tako da unatoč tome što je nepokretan postavlja izazov u izbjegavanju projektila.

To je implementirano uz pomoć skripte EnemyWeapon koja je duplicirana, tako da jedna verzija skripte ispaljuje jednu vrstu projektila određenom brzinom iz jednog rotirajućeg mjesta dok druga ispaljuje drugu vrstu projektila drugačijom brzinom iz mjesta koje se rotira drugačijom brzinom. Također ispod neprijatelja je instanca neprijatelja prvog tipa što onemogućava igraču da se spusti na pod, dakle potrebno je manevrirati u zraku i izbjegavati projektile neprijatelja kako bi se neprijatelj uništio.



Slika 10 Enemy4

4.6 Sustav uništavanja

Kako bi borba funkcionirala nužno je imati sistem uništavanja neprijatelja, te mogućnost neuspjeha, ili kako se to često naziva u video igrama – smrti. Zdravlje je univerzalan način prikazivanja ovog sistema, i u ovoj sekciji će biti prikazana njegova implementacija.

4.6.1 Kolizije

Svaka igra ovoga tipa daje način da se neprijatelj uništi, ali u isto vrijeme oni mogu uništiti i lika kojeg igrač kontrolira, što za igrača znači kraj igre (Game Over). To se obavlja u većini platformera na vrlo slične načine - ako igrač dođe u kontakt s neprijateljem on zadobiva udarac. Tako će biti i odrađeno u ovom projektu. Također ako neprijatelj dođe u kontakt s projektilom igrača on zadobiva udarac.

Unity u te i mnoge druge svrhe ima razvijenu metodu OnTrigger2D koja gleda kad collider jednog objekta dolazi u kontakt s colliderom drugog objekta. To može biti implementirano na neprijateljima da kad dođu kontakt s igračem oni njemu zadaju udarac, a može biti implementirano i na projektilu koji kad dođe u kontakt s neprijateljem njima zadaje udarac.

```
protected virtual void OnTriggerEnter2D(Collider2D collision)
{
    if(collision.gameObject == player)
    {
         hit = true;
         DealDamage(); // metoda objašnjena sljedeća
     }
}
```

4.6.2 Zdravlje

Nakon što i neprijatelji i naš lik detektiraju „udarce“ na njih potrebno je stvoriti sistem koji će ih nakon određenog broja „udaraca“ uništiti. To ide preko zdravlja, ili broja udaraca koje mogu primiti prije nego što se neprijateljski objekt uništi, ili u slučaju igrača prekine igra.

Prvo će biti stvorena klasa Health koju će nasljeđivati i neprijatelji i igrač, te sadrži osnovne podatke o količinama zdravlja, i o načinu oduzimanja zdravlja.

```
public class Health : Managers
{
    public int maxHealth;
    public int health;

    protected override void Initialization()
    {
        base.Initialization();
        health = maxHealth;
    }

    public virtual void DealDamage(int amount)
    {
        health -= amount;
    }
}
```

Metoda DealDamage koja za parametar traži integer varijablu oduzima od postojećeg zdravlja zadanu količinu te će se ovom metodom koristiti i neprijatelji i igrač. A sljedeće su klase stvorene posebno za Player objekt i neprijateljski objekt.

```
    protected virtual void DealDamage()
    {
        if(hit)
        {
            playerHealth.DealDamage(damageAmount);
            hit = false;
        }
  }
```

Klasa EnemyHealth se dakako odnosi na neprijatelja te se u njoj nalazi override funkcije DealDamage uz koju je u slučaju kolizije s projektilom igrača oduzima zdravlje te se nakon što se zdravlje spusti na nulu objekt prebacuje na neaktivan, dakle nestaje.

```
public override void DealDamage(int amount)
{
    base.DealDamage(amount);
    if(health <=0 && gameObject.GetComponent<EnemyCharacter>())
    {
        gameObject.SetActive(false);
        Invoke("WakyWaky", regenerateTime);
    }
}
```

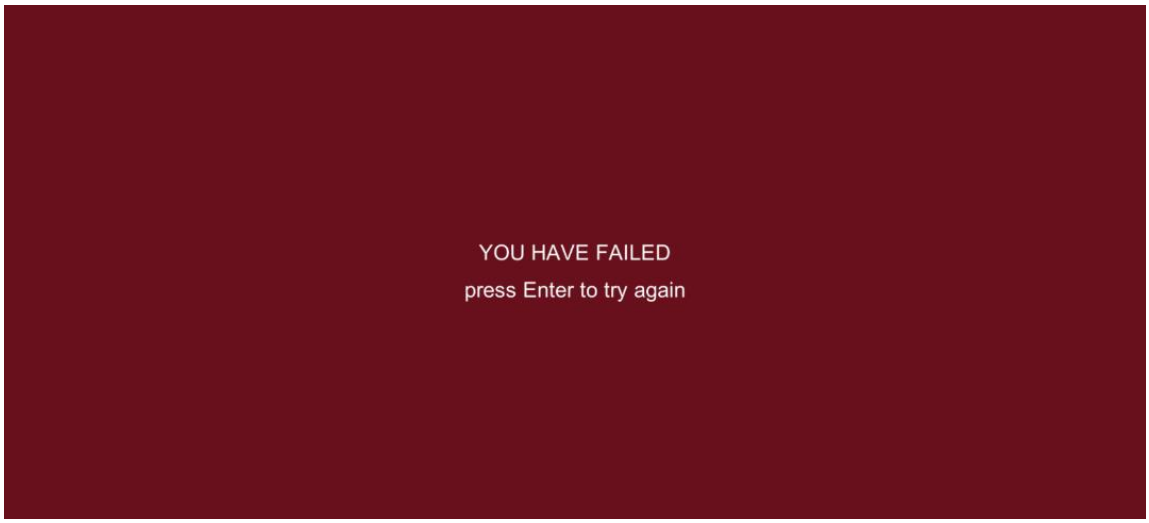
Uz to je ovdje zbog jednostavne strukture ovog prototipa dodana funkcija WakyWaky koja ponovno oživljuje neprijatelja koji je uništen nakon određenog broja sekundi jer nema drugačije izvedenog načina ponovnog susreta istih neprijatelja što je bitan dio svake Metroidvanie.

Klasa PlayerHealth sadrži malo više podataka. Osim premošćivanja DealDamage funkcije koja ima istu svrhu kao i u EnemyHealth tu je dodana i nemogućnost ponovnog zadavanja udarca određeno vrijeme (iFrameTime) nakon udarca (invulnerable). Uz to igrač dobije malu silu prema gore nakon što primi udarac kako bi se pokazao neki indikator za navedeno osim smanjenja zdravlja. Osim toga tu je naravno i funkcija koja se bavi događajem kad se igračevo zdravlje spusti na nulu – character.isDead.

```
protected virtual IEnumerator Died()
        {
            failScreen.SetActive(true);
            float timeStarted = Time.time;
            float timeSinceStarted = Time.time - timeStarted;
            float percentageComplete = timeSinceStarted / 2;
            Color currentColor = failScreenImage.color;
            Color currentTextColor = failScreenText.color;
            while(true)
            {
                timeSinceStarted = Time.time - timeStarted;
                percentageComplete = timeSinceStarted /2;
                currentColor.a = Mathf.Lerp(0, 1, percentageComplete);
```

```
            failScreenImage.color = currentColor;
            currentColor.a = Mathf.Lerp(0, 1, percentageComplete);
            failScreenText.color = currentTextColor;
            if(percentageComplete >= 1)
            {
                break;
            }
            yield return new WaitForEndOfFrame();
        }
    }
```

Ona postavlja na aktivno UI element prethodno stvoren u Unity Editoru – Game Over sliku, uz neke zanimljive efekte polaganog prelaska.



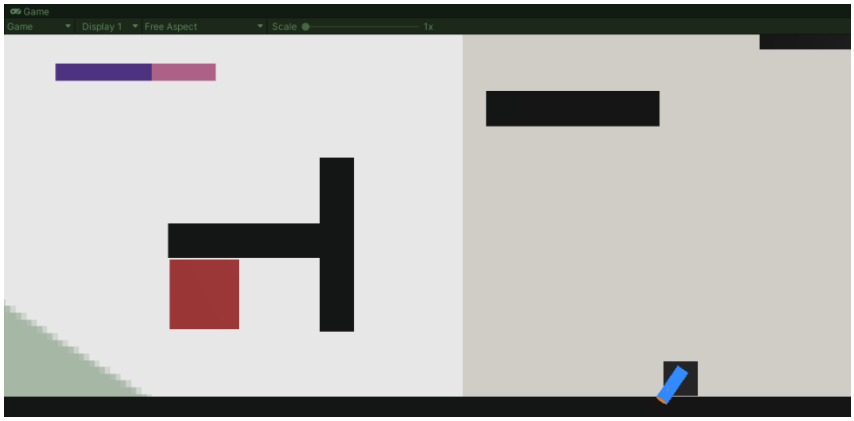
Slika 11 Game Over

Kao što vidimo, tipka Enter služi kako bi resetirali razinu. Način za to je napravljen na novom objektu Level Manager kojem je trenutno jedina svrha da resetira razinu nakon što je Enter pritisnut.

```
protected virtual void RestartLevel()
{
    SceneManager.LoadScene("SampleScene");
}
```

4.6.3 Pokazivač zdravlja

Uz pomoć UI objekata (user interface) u lijevi gornji kut ekrana su stavljeni dva pravokutnika jedan na drugi. Jedan služi kao pokazivač koliko zdravlja ima a drugi kao prazan prostor nakon što se prvi krene trošiti zadobivanjem udaraca. Trošenje zdravlja je implementirano preko komponente Slider koja je dodana na pravokutnik koji je stvoren. Preko koda također dodanog na objekt slider smanjuje i povećava postotak količine zdravlja koju lik posjeduje.



Slika 12 Pokazivač zdravlja u lijevom gornjem kutu

4.7 Input Manager i skupljanje sposobnosti

Posljednja skripta koja će biti stvorena se naziva InputManager, te kao što i ime govori služi za objedinjavanje svih tipki koje nešto čine. Uz nju imamo mogućnost dodjeljivanja tipke koje želimo staviti za određenu akciju u Inspectoru, umjesto samo u skripti.

Prvotna svrha je bila ovo što je prvo spomenuto, no s obzirom da ovo prototip Metroidvania igre, skripta se pokazala pogodnom i za drugu svrhu. S time da igrač ima mnogo sposobnosti kao u svakoj Metroidvaniji, nikako ne bi bilo u redu da ih posjeduje otpočetka. Tako da u pravom duhu Metroidvania igre, sposobnosti će se pokupiti tijekom igranja kratke prototip razine koja će biti stvorena na kraju. Tako će sposobnosti biti dostupne tek nakon prolaženja određenih izazova u obliku neprijatelja, a tek uz dobivanje novih sposobnosti će prelazak na druge dijelove razine biti moguće.

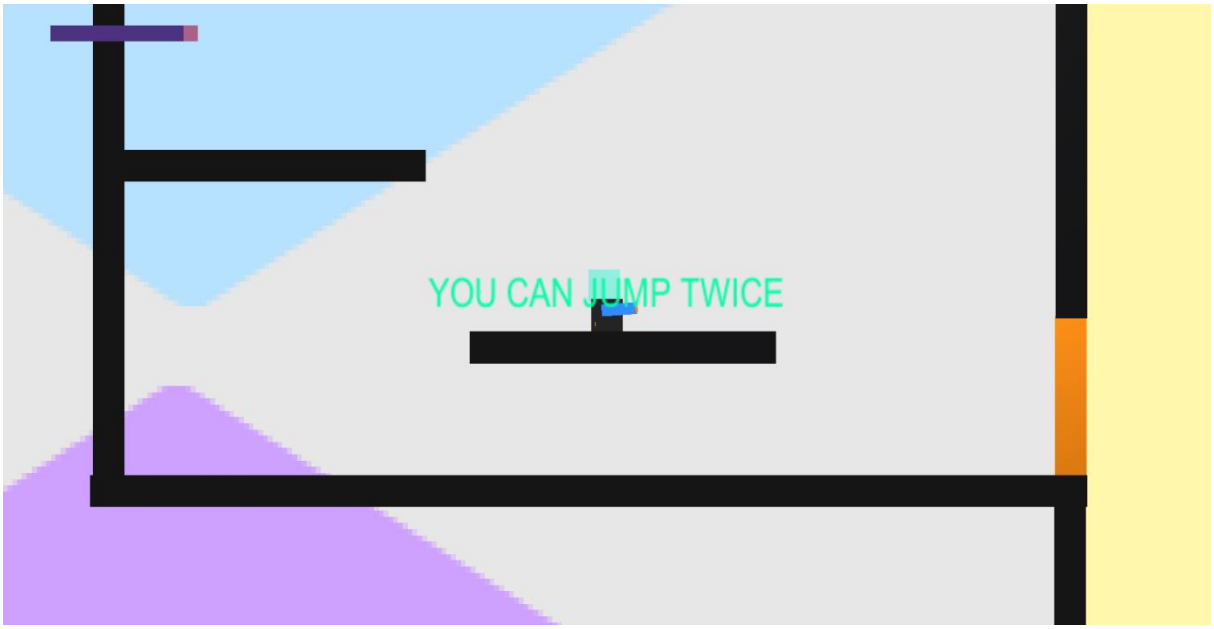
```
public bool SprintHeld()
{
    if(Input.GetKey(KeyCode.LeftShift) && sprintAbility)
    {
        return true;
    }
    else return false;
}
```

Input Manager skripti je zbog toga dodana i druga svrha da provjerava ako je određena sposobnost „skupljena“, a tek nakon toga bool koji je uvjet za korištenje te tipke u kontrolama postaje istinit, i na taj način lik primjerice može skakati dva puta.

„Skupljanje“ sposobnosti se odrađuje uz pomoć metode OnTrigger2D koja traži „tag“ objekta i aktivirana je onda kad igrač uđe u Collider objekta s nazivom sposobnosti. Nakon toga je bool postavljen na istinu te igrač dobiva novu sposobnost.

```
if(other.gameObject.CompareTag("Double Jump"))
{
    doubleJump = true;
    secondJump.SetActive(true);
}
```

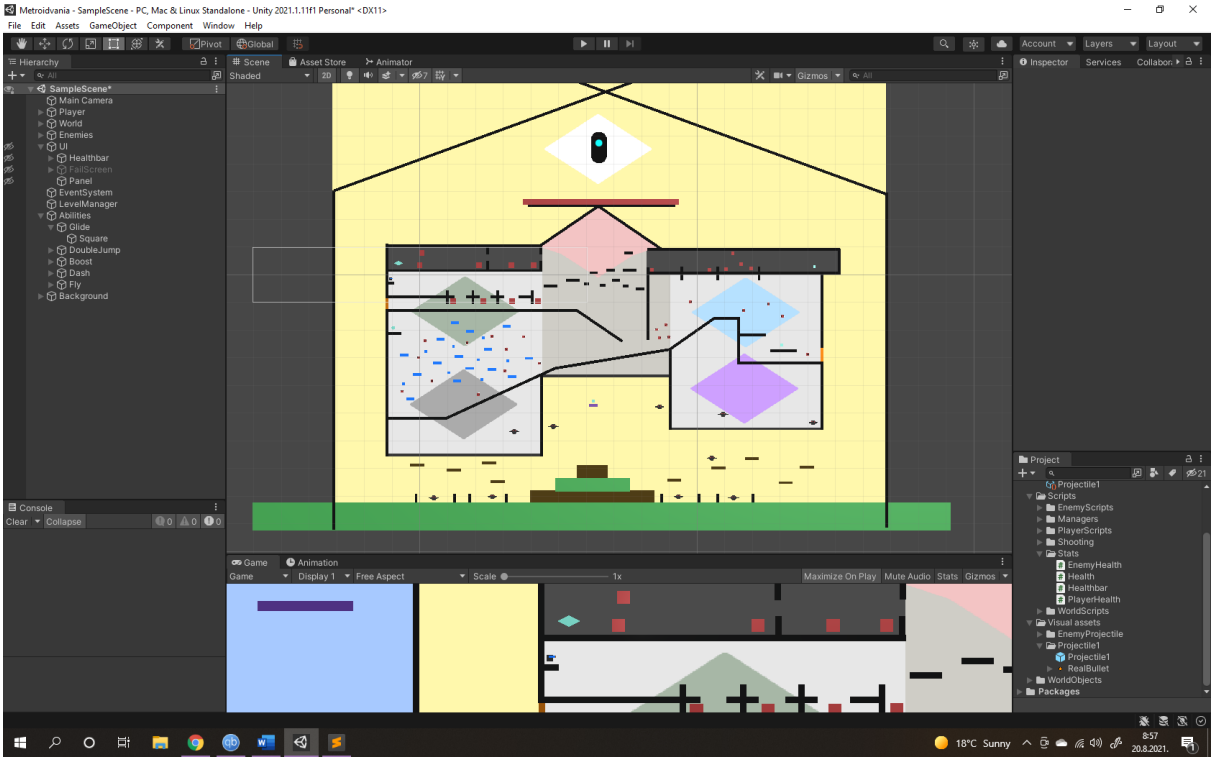
Sposobnosti su u svijetu stvorene kao 2D oblici svijetlo zelene boje, te uz njih je i aktiviran tekst koji se pojavi na slici u trenutku skupljanja koji govori o tipki koja je upravo omogućena, za što je vezana treća linija prikazanog isječka koda. Tekst nestaje nakon nekoliko sekundi jednostavnom metodom koja ponovno tekst prebacuje na neaktivan.



Slika 13 Dobivena nova sposobnost

4.8 Stvaranje prototip razine

Prototip igre je zamišljen kao minijatura prave, velike Metroidvania igre. Nekoliko različitih ali vrlo malih područja gdje se u svakom može pronaći posebna vrsta neprijatelja, objekti koji otključavaju nove sposobnosti na određenim dijelovima tih područja i svijet koji ima više putova kojima se može ići od kojih neki postaju dostupni tek uz novu sposobnost.



Slika 14 Završen dizajn razine

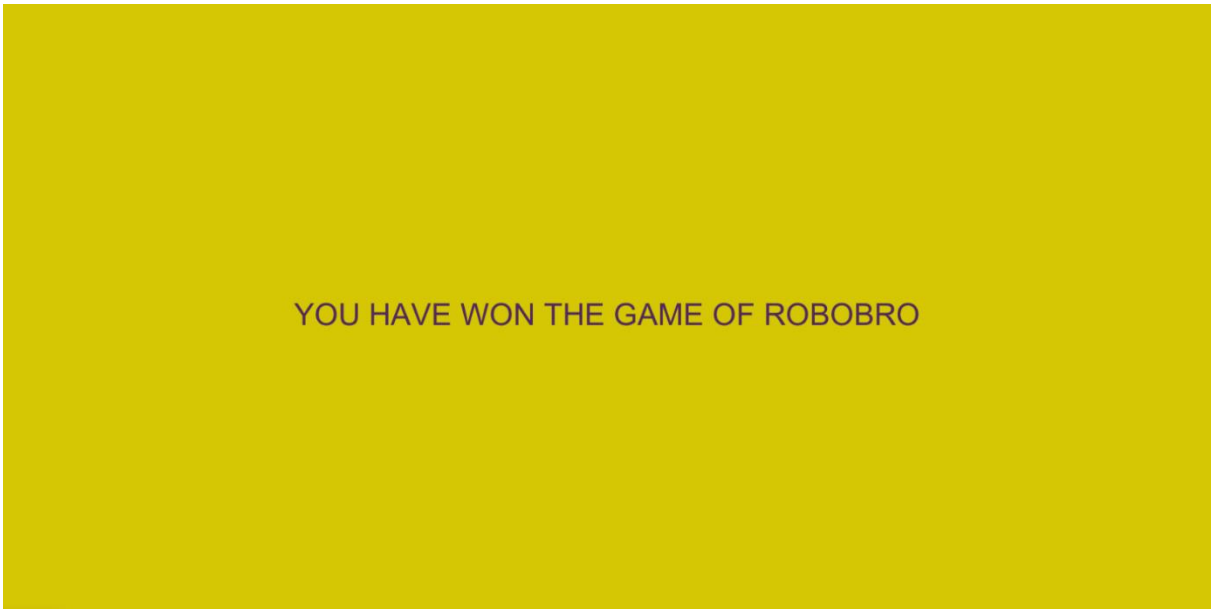
Igrač kreče bez mogućnosti za trčanjem (boost), lebdenjem, duplim skokom i dash na jednom dijelu svijeta gdje nema težih izazova. Nakon toga može nastaviti ravno te se uz malo skakutanja po platformama i izbjegavanja prvog tipa neprijatelja može domoći prve sposobnosti – lebdenje. Ako padne dolje prvo će morati skupiti druge sposobnosti kako bi se domogao lebdenja, jer još nema mogućnosti da skoči natrag.

Onda ima izbor otići lijevo ili desno gdje ga na obje strane čeka isti neprijatelj, ali ako ode lijevo ne može nikako doći do sposobnosti koja ga tamo čeka, tako da mora ići desno. Tamo pronalazi mogućnost dodatnog skoka, uz koji se sad može popeti natrag do lebdenja u slučaju da prije do nje nije uspio doći, a ako posjeduje i lebdenje i dupli skok sad može preko padajućih platforma doći i do treće sposobnosti – boost.

Uz ovu sposobnost sad jedino što je ostalo neistraženo je mjesto koje se vidi odmah po početku igre – dvije sobe iznad početnih komora sa kockama koje skaču na nepredvidive načine. Ako i tamo bude uspješan sad ima sposobnost dash s kojom može proći kroz narančasti zid s kojim se dosad vjerojatno susreo.

Prolaskom kroz navedeni zid igrač pada dolje te dolazi u novo područje novih neprijatelja koji zahtijevaju više projektila u sebi da bi bili uništeni, ali su zato sporiji i nezgrapniji. U središtu ovog područja u koje se može doći i s lijeve i s desne strane se nalazi pomična platforma koja igrača vodi do zadnje sposobnosti ovog prototipa – letenje.

Uz mogućnost letenja igrač može izbjegavati neprijatelje koji također lete te odletjeti visoko, iznad svih dosadašnjih dijelova i doći na krov kreirane građevine gdje se suočava s boss neprijateljem nakon čijeg poraza je igra pobijeđena.



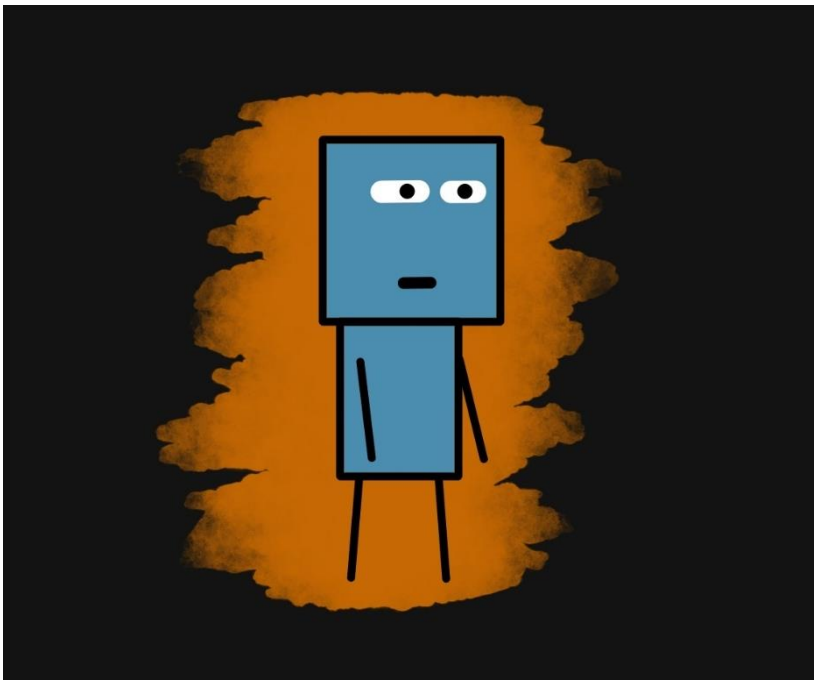
Slika 15 Pobjeda

4.9 Dodatna uređivanja

Nakon što je su sve mehanike, razine, kodovi, neprijatelji i sve što video igra mora sadržavati završeni, igranje igre ima smisla, ali nešto nedostaje. Igra se ne doima živo, već samo mlako. U tu svrhu je danas vrlo popularan pojam nazvan „game feel“, koji označava jednostavno rečeno koliko se u igru moguće uživjeti.

Tako da bi igra oživjela potrebna su mnoga dodatna uređivanja, koja se na prvi pogled čine manje relevantna, ali kad se dođe na ovu razinu završetka prototipa postaje očito kako su zapravo vjerojatno i bitnija od prošlo kreiranih aspekata igre. Možda najvažniji dio „game feel-a“ bi bili zvukovi koji funkcioniraju u skladu s igranjem igre. Zbog tog razloga su u Metroidvania igri kreiranoj ovdje dodani i zvukovi koji unatoč tome što ne odzvanjaju kvalitetom opet daju neki život igri.

Uz zvukove igre, drugi najvažniji aspekt uživljenosti bi se moglo reći da je glazba. Glazba je ono što vrlo često igranje igre pretvori iz solidnog iskustva u nezaboravno, zbog toga nije čudno koliko se pažnje posvećivalo glazbi u video igrama praktički svih generacija. Isto vrijedi i za ovu igru; tema ove igre je napisana u stilu tipičnog zvuka koji je odmah prepoznatljiv za video igre uz elemente modernije glazbe, što čini zanimljivu kombinaciju te veoma podiže iskustvo igranja igre kojoj je tijekom kreiranja nadjenuto ime Robobro.



Slika 16 Robobro zamišljena naslovna slika

5 Zaključak

S obzirom na to koliko je vremena i truda potrošeno na ovako jednostavan prototip već prethodno jasnih i korištenih mehanika, čini se da stvaranje video igara stvarno jest veoma težak zadatak. Postaje i znatno teži kad je sve na jednoj osobi kao u ovom slučaju, ali uz to je baš to način da za upoznavanjem svih dijelova „Game Development“ procesa te mogućnost za stvaranjem daleko većeg općenitog znanja o stvaranju video igara nego u slučaju podijele specifičnog zadatka onome koji ima znanja u potrebnom području. Ovo je prilika za konkretnim razumijevanjem mukotrpnosti ali i ljepota svih aspekata koji video igre uključuju.

Makar ovako izložen proces izgleda kao da je prošao glatko, ali to je daleko od istine. Konstantni bugovi, borbe s razumijevanjem funkcioniranja aplikacije, nedostatak znanja u C#, i još mnogo drugih su problemi koji su bili sveprisutni tijekom ovog projekta.

Zbog kreativnog dizajna svijeta veliki dio posla oko animacija, koji je inače nužan u video igrama je preskočen. Osim toga, neki dijelovi su naravno mogli biti usavršeniji, bugovi još malo bolje odrađeni, zvukovi bolji, ali sve su to detalji koji se vrlo lako mogu usavršiti tijekom ostatka procesa. Tako da primjerice uređivanje i postavljanje zvukova u Unity nisam shvatio najbolje, tako da su zvukovi koji jesu završili u igri postavljeni na najočitiji mogući način, te funkcioniraju dovoljno dobro. Ali zvukove neprijatelja koji bi trebali funkcionirati u prostoru nisam uspio postaviti.

Sve u svemu, Unity je puno pristupačniji nego što sam imao dojam prije početka rada u njemu, većina funkcija su odrađene vrlo logično i intuitivno. Shvaćanje pisanja koda i Unity metoda u C# je zahtijevalo malo više truda ali također nakon određenog vremena se jasno vidi temelj na kojem ono funkcionira te postaje intuitivno kao i svaki drugi dio programa.

Na početku ovaj projekt je izgledao neostvariv, ali jedan dan, nakon mnogo studiranja programa kroz razne upute i videa, rad u aplikaciji je za mene kliknuo. Nakon toga je postalo vrlo lako trošiti sate i sate na smišljanje logike jednostavnih kretnji poput duplog skoka ili lebdenja, a pogotovo sustav ispaljivanja projektila koji je zahtijevao malo višu razinu razumijevanja koda on one koju sam posjedovao. Bez obzira na to, u toj fazi je rad išao naprijed sam od sebe te je bez previše stanki odrađen do kraja.

6 Popis literature

1) Andrew Beatti (2021.), *How the Video Game Industry Is Changing* preuzeto, 26.8.2021.- https://www.investopedia.com/articles/investing/053115/how-video-game-industry-changing.asp
2) J. Clement (2021.), *Video game industry - Statistics & Facts*, preuzeto 25.8.2021. https://www.statista.com/topics/868/video-games/
3) MomentsOnline (2020.), *Brief History of Game Engine Then Till Now*, preuzeto 25.8.2021. - https://momentsonline69.medium.com/create-any-game-with-gdevelop-game-engine-9c8ca2344921
4) Dawn Spring (2014.), *Gaming history: computer and video games as historical scholarship*, preuzeto 25.8.2021. - https://www.tandfonline.com/doi/full/10.1080/13642529.2014.973714
5) Eric Peckham (2019.), *How Unity Built the Worlds Biggest Game Engine,* preuzeto 26.8.2021. – https://techcrunch.com/2019/10/17/how-unity-built-the-worlds-most-popular-game-engine/?guccounter=1&guce_referrer=aHR0cHM6Ly93d3cuZ29vZ2xlLmNvbS8&guce_referrer_sig=AQAAABoFIDTXNv4LsToHpCa23Cd0iiYPyrsTchSLHlDF4qQ49wR27tzSaUM8WI8AFt15ks64CwxlAT6Q0jRiF5iXNav_sUYwQ8LDz8xvvz-9I-KSmdc3IWIDTC8hOEaXNh7OwjuFX7gn0lCyOqMX5BlYiRWjvuMX5wfNHOvZd8xVYyvl
6) Unity podaci o verzijama, preuzeto 5.8.2021. - https://store.unity.com/compare-plans
7) Wikipedia (2021.), *Metroidvania*, preuzeto 25.8.2021. - https://en.wikipedia.org/wiki/Metroidvania#:~:text=Metroidvania%20is%20a%20subgenre%20of,genre%20borrowing%20from%20both%20series.
8) Mengling Wu (2017.), *Indie Game Industry Case Study*, preuzeto 24.8.2021. - https://repository.library.northeastern.edu/files/neu:cj82rk43p/fulltext.pdf

7 Tehnička dokumentacija

1) GetAxis - https://docs.unity3d.com/ScriptReference/Input.GetAxis.html
2) GetKey - https://docs.unity3d.com/ScriptReference/Input.GetKey.html
3) AddForce - https://docs.unity3d.com/ScriptReference/Rigidbody.AddForce.html
4) Rigidbody - https://docs.unity3d.com/ScriptReference/Rigidbody.html
5) Collider – https://docs.unity3d.com/ScriptReference/BoxCollider2D.html
6) Game Object - https://docs.unity3d.com/Manual/GameObjects.html
7) Komponente objekta - https://docs.unity3d.com/Manual/Components.html
8) GameObject Transform - https://docs.unity3d.com/Manual/class-Transform.html
9) Tagovi - https://docs.unity3d.com/Manual/Tags.html
10) Mathf.Clamp - https://docs.unity3d.com/ScriptReference/Mathf.Clamp.html
11) RaycastHit2D - https://docs.unity3d.com/ScriptReference/Physics2D.Raycast.html
12) Dash sposobnost - https://generalistprogrammer.com/unity/unity-2d-dash-movement-effect-learn-to-how-to-tutorial/
13) Prefabs - https://docs.unity3d.com/Manual/CreatingPrefabs.html
14) Vector2 - https://docs.unity3d.com/ScriptReference/Vector2.html
15) Vector3 - https://docs.unity3d.com/ScriptReference/Vector3.html
16) Uvod u object pooling-
17) https://learn.unity.com/tutorial/introduction-to-object-pooling#5ff8d015edbc2a002063971d
18) Skriptirani objekti - https://docs.unity3d.com/Manual/class-ScriptableObject.html
19) User Interface - https://docs.unity3d.com/Manual/UIToolkits.html
20) Audio - https://docs.unity3d.com/Manual/Audio.html
21) Physics2D - https://docs.unity3d.com/Manual/class-Physics2DManager.html
22) Stvaranje scena - https://docs.unity3d.com/Manual/scenes-working-with.html
23) Detekcija kolizija - https://docs.unity3d.com/Manual/LayerBasedCollision.html
24) Orijentacija - https://docs.unity3d.com/Manual/QuaternionAndEulerRotationsInUnity.html

25) Skripte - https://docs.unity3d.com/Manual/CreatingAndUsingScripts.html
26) Unity Klase- https://docs.unity3d.com/Manual/ScriptingImportantClasses.html
27) Coroutines - https://docs.unity3d.com/Manual/Coroutines.html

8 Popis slika